Concepts of Parallel and Distributed Database Systems

Key concepts:

- concept and structure of parallel and distributed databases
- motivations for parallelism and distribution
- types of parallelism
- benefits of DDBMS
- drawbacks of DDBMS
- Date’s rules
- local autonomy
- function distribution
- continues operation
- location independence
- fragmentation independence
- replication independence
- overview of query processing
- overview of distributed transaction management
- hardware independence
- DBMS independence
- Distribution transparency
- Replication transparency
- Fragmentation transparency
- history of parallel databases
- history of distributed databases

Parallel and Distributed Databases

- A parallel database aims principally linear speedup and scaleup:

  Linear scaleup refers to a sustained performance for a linear increase both in database size and processing and storage power.
  Linear speedup refers to a linear increase in performance for a constant database size.

- Sharing data is the key of a distributed DB → cooperative approach

  Since data is distributed, users that share that data can have it placed at the site they work on, with local control (Local autonomy)

- Distributed and parallel databases improve reliability and availability

  I.e. they have replicated components and thus eliminate single points of failure.

- The distribution of data and the parallel/distributed processing is not visible to the users → transparency

- Distributed Database (DDB)

A collection of multiple, *logically interrelated* databases *distributed over a computer network*

- Distributed Database Management System (D-DBMS)

  The software system that permits the management of the DDB and provides an access mechanism that makes this distribution transparent to the users.

- Distributed Database System (DDBS)

  DDBS = DDB + D-DBMS

- What is not a DDBS: A database system which resides on
  - a timesharing computer system
  - a loosely or tightly coupled multiprocessor system
  - one of the nodes of a network of computers - this is a centralised database on a network node

<u>Reasons of data distribution</u>

- Improved reliability and availability through distributed transactions
- Improved performance

- Allowing data sharing while maintaining some measure of local control
- Easier and more economical system expansion
- Distributed nature of some database applications

Additional functionality of a D-DBMS

- Distribution leads to increased complexity in the system design and implementation
- D-DBMS must be able to provide some additional functionality to those of a central DBMS:
    - To access remote sites and transmit queries and data among the various sites via a communication network.
    - To keep track of the data distribution and replication in the DDBMS catalog.
    - To devise execution strategies for queries and transactions that access data from more than one site.
    - To decide on which copy of a replicated data item to access.
    - To maintain the consistency of copies of a replicated data item.
    - To maintain the global conceptual schema of the distributed database.
    - To recover from individual site crashes and from new types of failures such as failure of a communication link.

Distributed DBMS Issues

- Distributed Database Design

- how to distribute the database
- replicated & non-replicated database distribution
- a related problem in directory management

- Query Processing
  - convert user transactions to data manipulation instructions
  - optimisation problem
  - min{cost = data transmission + local processing}
  - general formulation is NP-hard

- Concurrency Control
  - synchronisation of concurrent accesses
  - consistency and isolation of transactions' effects
  - deadlock management

- Reliability
  - how to make the system resilient to failures
  - atomicity and durability

Date’s 12 rules for distributed database systems

Rule 0: to the user, a distributed system should look exactly like a non-distributed system

other rules:

1. Local autonomy
2. No reliance on central site
3. Continuous operation
4. Location independence
5. Fragmentation independence
6. Replication independence
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network independence
12. DBMS independence

The great principles are:

Autonomy, Independence and Transparency.

Rule 1: Local Autonomy

Autonomy objective: Sites should be autonomous to the maximum extent possible.

- Local data is locally owned and managed, with local accountability
  - security considerations
  - integrity considerations
- Local operations remain purely local
- All operations at a given site are controlled by that site; no site X should depend on some other site Y for its successful functioning

  Otherwise, if site Y is down, site X could not carry out the operation although there is nothing wrong at site X.
- In some situations some slight loss of autonomy is inevitable
  - fragmentation problem – rule 5
  - replication problem – rule 6
  - update of replicated relation – rule 6
  - multiple-site integrity constraint problem – rule 7
  - a problem of participation in a two-phase commit process – rule 8

In consequence: All data ‘really’ belongs to some local database even it is accessible from remote.

Rule 2: No reliance on central site

> There must not be any reliance on a central ‘master’ site for some central service, such as centralised query processing or centralised transaction management, such that the entire system is dependent on that central site.

- Reliance on a central site would be undesirable for at least the following two reasons:
  - that central site might be a bottleneck
  - the system would be vulnerable
- In a distributed system therefore, the following functions (among others) must all be distributed:
  - Dictionary management
  - Query processing
  - Concurrency control
  - Recovery control

Rule 3: Continuous operation

There should ideally never be any need for a planned entire system shutdown .

- Incorporating a new site into an existing distributed system should not bring the entire system to a halt
- Incorporating a new site into an existing distributed system should not require any changes to existing user programs or terminal activities
- Removing an existing site from the distributed systems should not cause any unnecessary interruptions in service
- Within the distributed system, it should be possible to create and destroy fragments and replicas of fragments dynamically
- It should be possible to upgrade the DBMS at any given component site to a newer release without taking the entire system down
-

Rule 4: Location independence (Transparency)

Users should not have to know where data is physically stored but rather should be able to behave – at least from a logical standpoint – as if the data was all stored at their own local site.

- Sometimes referenced as *Transparency*
- Simplifies user programs and terminal activities
- Allows data to migrate from site to site
- It is easier to provide location independence for simple retrieval operations than it is for update operations
- Distributed data naming scheme and corresponding support from the dictionary subsystem
- User naming scheme
  - An user has to have a valid logon ID at each of multiple sites to operate
  - User profile for each valid logon ID in the dictionary
  - Granting of access privileges at each component site
-

<u>Rule 5: Fragmentation independence (Transparency)</u>

- A distributed system supports *data fragmentation* if a given relation can be divided up into pieces or 'fragments' for physical storage purposes
- A system that supports data fragmentation should also support *fragmentation independence* (also known as fragmentation transparency)

> Users should be able to behave (at least from a logical standpoint) as if the data were in fact not fragmented at all.

- Fragmentation is desirable for performance reasons
- Fragmentation must be defined within the context of a distributed database
- Fragmentation independence (like location independence) is desirable because it simplifies user programs and terminal activities
- Fragmentation independence implies that users should normally be presented with a view of the data in which the fragments are logically combined together by means of suitable joins and unions

Fragmentation principles

- Two basic kinds of fragmentation
  - Horizontal fragmentation
    - refers to cut between tuples
    - achieved by *selection*
    - same as data partitioning in parallel databases
    - efficient for parallel scanning of the relation
  - Vertical fragmentation
    - refers to the cut of the schema

  - achieved by a *projection*
  - efficient if there is frequent access to different attribute groups

- The *union* of the horizontal fragments should be the original relation

- The *join* of the vertical fragments should be the original relation

  !!! Supplemental measure must be undertaken to guarantee that the join is nonloss!

  E.g. the fragmentation of the table
  Employee(Emp#, Dept#, Salary) into the fragments
  Emp#, Dept# and *Salary* would not be valid, since there is no common join-attribute.

  Solutions are either to duplicate the key or to introduce a hidden ‘tupleID’ where the TID is the physical or logical address for that tuple. This TID is then included in all fragments.

Rule 6: Replication independence (Transparency)

User should be able to behave as if the data were in fact not replicated at all.

- A distributed system supports data replication if a given relation (more generally, a given fragments of a relation) can be represented at the physical level by *many distinct stored copies* or *replicas*, at many distinct sites.
- Replication, like fragmentation, should be “transparent to the user”
- Replication is desirable for at least two reasons:
  - Performance: applications can operate on local copies instead of having communicate with remote sites (or a ‘nearest’ copy can be fetched.
  - Availability: a given replicated object remains available for processing as long as at least one copy remains available.
- Update propagation problem!
- Fragmentation independence (like all transparency rules) is desirable because it simplifies user programs and terminal activities

Update propagation problem

- The major disadvantage of replication is, of course, when a given replicated object is updated, all copies of that object must be updated.

→ Update propagation problem.

- A common scheme for dealing with this problem is the so-called primary copy scheme:
  - One copy of each replicated object is designated as the *primary copy*. The remainder are all secondary copies.
  - *Primary copies* of different objects are at different sites (so the distributed scheme once again).
  - Update operations are deemed logically complete as soon as the primary copy has been update. The site holding the copy is then responsible for propagating the update to the secondary copies at some subsequent time.
- The 'subsequent' time must be prior to COMMIT, however, if the ACID properties of the transaction are to be preserved $\rightarrow$

  Inconsistency ?! (see below: transaction processing).

Rule 7: Distributed query processing

It is crucially important for distributed database systems to choose a good strategy for distributed query processing.

- Query processing in a distributed system involve
  - local CPU and I/O activity at several distinct sites
  - some amount of data communication among those sites

- Amount of data communication is a *major* performance factor. Depending on the bandwidth and the relation sizes *the minimal communication cost strategy* will be chosen.
- Distributed query processing and query optimisation must be aware of the distributed context:
  - Where is a relation (fragment) stored?
  - Are there any replicas?
- Query compilation ahead of time
- Views that span multiple sites
- Integrity constraints within a DDBS that span multiple sites

Rule 8: Distributed transaction management

Two major aspects of transaction management**,** recovery control and concurrency control, require extended treatment in the distributed environment

- In a distributed system, a single transaction can involve the execution of code at multiple sites and can thus involve updates at multiple sites

- Each transaction is therefore said to consist of multiple “agents”, where an agent is the process performed on behalf of a given transaction at a given site
- Recovery Control**:** In order to ensure that a given transaction is atomic in the distributed environment, therefore, the system must ensure that the set of agents for that transaction either all commit in unison or all roll back in unison.

  That effect can be achieved by means of the two-phase commit protocol.
- Concurrency Control**:** is based in most distributed systems on locking (some implemented multi-version control ..., read can have the last version)

Recovery control: Two-phase commit protocol

- In general, the two-phase commit protocol is driven, whenever a transaction interact with several independent 'resource manager', i.e. each managing its own set of recoverable resources and maintaining its own recovery log.
- FIRST PHASE**:** Each site must force-write all log entries for local resources used by the transaction out to its physical log. Assuming the force-write is successful, each site gives the OK.

- SECOND PHASE**:** If the site where the transaction has been submitted receives all the OK form the other sites, it force-writes an entry to its own physical log.

  If anything has gone noOK, then general ROLLBACK otherwise the decision is COMMIT DONE.

- *Remark 1:* If site Y acts as a participant in a two-phase commit coordinated by site X, then Y must do what is told by site X (roll-back) – a (minor) loss of local autonomy.

- *Remark 2:* If one of the messages get lost (e.g. ROLLBACK) then the system is an inconsistent state. In general there can be no protocol that can guarantee that all agents will commit unison.

Concurrency control: Primary Copy approach

- Lock means: Requests to test, set and release locks. Update means update and acknowledgement.

- Consider for example a transaction that needs to update an object for which exists replicas at n sites. A straightforward implementation will require:

  n lock requests; *n* lock grants; *n* update messages; *n* acknowledgements; n unlock requests

  makes 5**n* messages (could of course be piggy-picked)

- Use the primary copy: Lock only the primary copy (one lock request, one lock grant and one unlock) and the site with the primary copy handles the updating (2**n* messages).

  Thus we got only 2**n*+3 messages.

- *Remark 1:* Loss of autonomy**:** A transaction can fail now if a primary copy is unavailable, and a secondary copy exists at the local site.

- *Remark 2:* Inconsistency is introduced if all secondary copies are not updated prior to the COMMIT point.

  Unfortunately, many commercial products support a less ambitious form of replication in which the update propagation is guaranteed to be done at some future time. They put the term replication (performance) against two-phase commit protocol.

Rule 9-11: Hardware, OS and Network Independence

User should be presented with the "single-system image" regardless any particular hardware platform, operating system and network.

- It is desirable to be able to run the same DBMS on different hardware systems which all participate as equal partners (where appropriate) in a distr. system.

The strict homogeneity assumption is not relaxed; it is still assumed that the same DBMS is running on all those different hardware systems.

- From a commercial point of view, the most important operating system environments, and hence the ones that (at a minimum) the DBMS should support, are probably MVS/XA, MVS/ESA, VM/CMS, VAX/VMS, UNIX (various flavours), OS/2, MS-DOS, Windows

- From the D-DBMS viewpoint, the network is merely the provider of a reliable message transmission service.

  By "reliable" here is meant that, if the network accepts a message from site X for delivery to site Y, then it will eventually deliver that message to site Y.

  Messages will not be garbled, will not be delivered more than once and will be delivered in the order sent.

  The network should also be responsible for site authentication.

Rule 12: DBMS independence

Ideal distributed system should provide DVBMS independence (or transparency).

- The obvious fact is that not only the real-world computer installations typically run on many different machines and with many different operating systems, but very often run with different DBMSs as well.
- All that is really needed is that the DBMS instances at different sites all support the same interface**.**
- Consider an example: Suppose that site *X* is running INGRES and site *Y* is running ORACLE. Some user at site *X* desires to see a single distributed database that includes data from INGRES (site *X*) and ORACLE from site *Y*. By definition user *U* is a an INGRES user and the distributed database must therefore be an INGRES database.
- The *solution* is quite straightforward: INGRES must provide an application program (gateway) that runs upon ORACLE and has the effect of 'making ORACLE look like INGRES'.
- Implementing protocols for the exchange of information between INGRES and ORACLE involves first the understanding of the messages in which SQL sources are sent form INGRES and translated to ORACLE statements and mapping ORACLE results (data values, return codes, ...) into the messages format that INGRES expects.

Types of parallelism in database systems

- Interquery parallelism

  Multiple queries generated by concurrent transactions can be executed parallel.

- Interoperation parallelism

  A query may consist of multiple, independent operations that can be executed parallel. There are two forms of interoperation parallelism:

- Independent
- Pipeline

- Intraoperation parallelism

  An operator may be executed as many small, independent sub-operations. The relational model is ideal for implementing intraoperator parallelism.

  This is the most important type of parallelism, since generally there are much more data tuples than relational operations.

  Intraoperation parallelism comes from data partitioning.

Types of parallelism in database systems